Delayed evaluation

For each language construct, the semantics specifies when subexpressions get evaluated. In ML, Racket, Java, C:

- Function arguments are **eager** (call-by-value)
  - Evaluated once before calling the function
- Conditional branches are not eager

It matters: calling `factorial-bad` never terminates:

```scheme
(define (my-if-bad x y z)
  (if x y z))

(define (factorial-bad n)
  (my-if-bad (= n 0)
    1
    (* n (factorial-bad (- n 1)))))
```
Thunks delay

We know how to delay evaluation: put expression in a function!
   - Thanks to closures, can use all the same variables later

A zero-argument function used to delay evaluation is called a thunk
   - As a verb: thunk the expression

This works (but it is silly to wrap if like this):

```
(define (my-if x y z)
  (if x (y) (z)))

(define (fact n)
  (my-if (= n 0)
    (lambda() 1)
    (lambda() (* n (fact (- n 1))))))
```
The key point

- Evaluate an expression $e$ to get a result:
  $$e$$

- A function that *when called*, evaluates $e$ and returns result
  - Zero-argument function for “thunking”
    $$\texttt{(lambda () e)}$$

- Evaluate $e$ to some thunk and then call the thunk
  $$\texttt{(e)}$$

- Next: Powerful idioms related to delaying evaluation and/or avoided repeated or unnecessary computations
  - Some idioms also use mutation in encapsulated ways
Avoiding expensive computations

Thunks let you skip expensive computations if they are not needed

Great if take the true-branch:

```
(define (f th)
  (if (…) 0 (... (th) ...)))
```

But worse if you end up using the thunk more than once:

```
(define (f th)
  (... (if (…) 0 (... (th) ...))
       (if (…) 0 (... (th) ...))
       ...
       (if (…) 0 (... (th) ...))))
```

In general, might not know many times a result is needed
**Best of both worlds**

Assuming some expensive computation has no side effects, ideally we would:

- Not compute it *until needed*
- *Remember the answer* so future uses complete immediately

Called *lazy evaluation*

Languages where most constructs, including function arguments, work this way are *lazy languages*

- Haskell

Racket predefines support for *promises*, but we can make our own

- Thunks and mutable pairs are enough
Delay and force

```
(define (my-delay th)
  (mcons #f th))

(define (my-force p)
  (if (mcar p)
      (mcdr p)
      (begin (set-mcar! p #t)
              (set-mcdr! p ((mcdr p)))
              (mcdr p))))
```

An ADT represented by a mutable pair

- #f in car means cdr is unevaluated thunk
  - Really a one-of type: thunk or result-of-thunk
- Ideally hide representation in a module
Using promises

```
(define (f p)
  (... (if (...) 0 (... (my-force p) ...))
       (if (...) 0 (... (my-force p) ...)))
  ...
  (if (...) 0 (... (my-force p) ...)))

(f (my-delay (lambda () e)))
```
Lessons From Example

See code file for example that does multiplication using a very slow addition helper function

• With thunking second argument:
  – *Great* if first argument 0
  – Okay if first argument 1
  – *Worse* otherwise

• With precomputing second argument:
  – *Okay* in all cases

• With thunk that uses a promise for second argument:
  – *Great* if first argument 0
  – *Okay* otherwise
### Streams

- A stream is an *infinite sequence* of values
  - So cannot make a stream by making all the values
  - Key idea: Use a thunk to delay creating most of the sequence
  - Just a programming idiom

A powerful concept for division of labor:
- Stream producer knows how to create any number of values
- Stream consumer decides how many values to ask for

Some examples of streams you might (not) be familiar with:
- User actions (mouse clicks, etc.)
- UNIX pipes: `cmd1 | cmd2` has `cmd2` “pull” data from `cmd1`
- Output values from a sequential feedback circuit
Using streams

We will represent streams using pairs and thunks

Let a stream be a thunk that *when called* returns a pair:

'(next-answer . next-thunk)

So given a stream \( s \), the client can get any number of elements

- First: \((\text{car } (s))\)
- Second: \((\text{car } ((\text{cdr } (s))))\)
- Third: \((\text{car } ((\text{cdr } ((\text{cdr } (s))]))))\)

(Usually bind \((\text{cdr } (s))\) to a variable or pass to a recursive function)
Example using streams

This function returns how many stream elements it takes to find one for which tester does not return \#f

- Happens to be written with a tail-recursive helper function

```
(define (number-until stream tester)
  (letrec ([f (lambda (stream ans)
               (let ([pr (stream)])
                 (if (tester (car pr))
                     ans
                     (f (cdr pr) (+ ans 1))))])
    (f stream 1)))
```

- `(stream)` generates the pair
- So recursively pass `(cdr pr)`, the thunk for the rest of the infinite sequence
Streams

Coding up a stream in your program is easy
  – We will do functional streams using pairs and thunks

Let a stream be a thunk that \textit{when called} returns a pair:

\begin{verbatim}
'(next-answer . next-thunk)
\end{verbatim}

Saw how to use them, now how to make them…
  – Admittedly mind-bending, but uses what we know
Making streams

• How can one thunk create the right next thunk? Recursion!
  – Make a thunk that produces a pair where cdr is next thunk
  – A recursive function can return a thunk where recursive call does not happen until thunk is called

```
(define ones (lambda () (cons 1 ones)))
(define nats
  (letrec ([f (lambda (x)
                (cons x (lambda () (f (+ x 1))))))]
                  (lambda () (f 1)))))
(define powers-of-two
  (letrec ([f (lambda (x)
               (cons x (lambda () (f (* x 2)))))]
               (lambda () (f 2)))))
```
**Getting it wrong**

- This uses a variable before it is defined
  
  ```lisp
  (define ones-really-bad (cons 1 ones-really-bad))
  ```

- This goes into an infinite loop making an infinite-length list
  
  ```lisp
  (define ones-bad (lambda () cons 1 (ones-bad)))
  (define (ones-bad) (cons 1 (ones-bad)))
  ```

- This is a stream: thunk that returns a pair with cdr a thunk
  
  ```lisp
  (define ones (lambda () (cons 1 ones)))
  (define (ones) (cons 1 ones))
  ```
Memoization

• If a function has no side effects and does not read mutable memory, no point in computing it twice for the same arguments
  – Can keep a cache of previous results
  – Net win if (1) maintaining cache is cheaper than recomputing and (2) cached results are reused

• Similar to promises, but if the function takes arguments, then there are multiple “previous results”

• For recursive functions, this memoization can lead to exponentially faster programs
  – Related to algorithmic technique of dynamic programming
How to do memoization: see example

• Need a (mutable) cache that all calls using the cache share
  – So must be defined outside the function(s) using it

• See code for an example with Fibonacci numbers
  – Good demonstration of the idea because it is short, but, as shown in the code, there are also easier less-general ways to make `fibonacci` efficient
  – (An association list (list of pairs) is a simple but sub-optimal data structure for a cache; okay for our example)
**assoc**

- Example uses `assoc`, which is just a library function you could look up in the Racket reference manual:

  \[(\text{assoc } v \text{ lst})\] takes a list of pairs and locates the first element of `lst` whose car is equal to `v` according to `isequal?`. If such an element exists, the pair (i.e., an element of `lst`) is returned. Otherwise, the result is `#f`.

- Returns `#f` for not found to distinguish from finding a pair with `#f` in `cdr`